```
B = SUM_SCATTER(ARRAY = A, BASE = B, INDX1 = V)
```

Note the use of Fortran 90 keyword arguments (which can occur in any order). The result of any of the scatter functions has the same shape as its `BASE` argument. The value of an element of the result is the corresponding element of the base unless it is associated with a set of elements of `ARRAY`. Element k of the result is associated with and (in the case of `SUM_SCATTER` is set to the sum of the elements `ARRAY(i)` for which `V(i)` is equal to k.

9.2.3 Extended data parallelism

Is data parallelism the same as array-based parallelism? Some think so, but this is a misconception. The key idea is the application of a function simultaneously to a collection of arguments. HPF allows one to specify this in a very flexible way with the `INDEPENDENT DO` loop

```
!HPF$ INDEPENDENT
   DO I = 1, 1024
   ...
   ENDDO
```

The independent directive does not change what is computed. It just advises the compiler that the loop iterations may be executed in parallel, with no contention for or race conditions involving shared variables. When the use of some variable by multiple iterations of a loop renders this false, we can still use the INDEPENDENT directive, as long as the use of the variable is essentially local to each iteration and it does not carry information from one iteration to another. For example,

```
DO I = 1, N
   DO J = 1, 3
   ...
   ENDDO
ENDDO
```

Ordinarily, the variable `J` is not used to carry information between iterations of the I loop. We can assert that the outer loop is independent *if* the variable `J` is made local to each outer iteration by use of the directive

```
!HPF$ INDEPENDENT, NEW J
```

just before the `DO I` statement; this has the same effect as replacing the shared scalar `J` with an array, and using `J(I)` at iteration `I`, but it uses less storage.

9.3 Message Passing

In the SISD, SIMD, and MIMD computer taxonomy, SISD machines are conventional uniprocessors, SIMD are single instruction stream machines that operate in parallel on ensembles of data, like arrays or vectors, and MIMD machines have multiple active threads of control (processes) that can operate on multiple data in parallel. How do these threads share data and how do they synchronize? For example, suppose two processes have a producer/consumer relationship. The producer generates a sequence of data items that are passed to the consumer, which processes them further. How does the producer deliver the data to the consumer?

If they share a common memory, then they agree on a location to be used for the transfer. In addition, they have a mechanism that allows the consumer to know when that location is full, *i.e.* it has a valid datum placed there for it by the producer, and a mechanism to read that location and change its state to empty. A full/empty bit is often associated with the location for this purpose. The hardware feature that is often used to do this is a "test-and-set" instruction that tests a bit in memory and simultaneously sets it to one. The producer has the obvious dual mechanisms.

Many highly parallel machines have been, and still are, just collections of independent computers on some sort of a network. Such machines can be made to have just about any data sharing and synchronization mechanism; it just depends on what software is provided by the operating system, the compilers, and the runtime libraries. One possibility, the one used by the first of these machines (The Caltech Cosmic Cube, from around 1984) is message passing. (So it's misleading to call these "message passing machines"; they are really multicomputers with message passing library software.)

From the point of view of the application, these computers can send a message to another computer and can receive such messages off the network. Thus, a process cannot touch any data other than what is in its own, private memory. The way it communicates is to send messages to and receive messages from other processes. Synchronization happens as part of the process, by virtue of the fact that both the sending and receiving process have to make a call to the system in order to move the data: the sender won't call send until its data is already in the send buffer, and the receiver calls receive when its receive buffer is empty and it needs more data to proceed.

Message passing systems have been around since the Cosmic Cube, about ten years. In that time, there has been a lot of evolution, improved efficiency, better software engineering, improved functionality. Many variants were developed by users, computer vendors, and independent software companies. Finally, in 1993, a standardization effort was attempted, and the result is the Message Passing Interface (MPI) standard. MPI is flexible and general, has good implementations on all the machines one is likely to use, and is almost certain to be around for quite some time. We'll use MPI in the course.

In print, the best MPI reference is the handbook *Using MPI*, by William Gropp, Ewing Lusk, and Anthony Skjellum, published by MIT Press ISBN 0-262-57104-8.

The standard is on the World WIde Web. The URL is
http://www.mcs.anl.gov/mpi/mpi-report/mpi-report.html. An updated version is at
ftp://ftp.mcs.anl.gov/pub/mpi/mpi-1.jun95/mpi-report.ps

9.3.1 Sending and receiving

In order to get started, let's look at the two most important MPI functions, `MPI_Send` and `MPI_Recv`.

The call

```
MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
         int tag, MPI_Comm comm)
```

sends `count` items of data of type `datatype` starting at the location `buf`. In all message passing systems, the processes have identifiers of some kind. In MPI, the process is identified by its `rank`, an integer. The data is sent to the processes whose rank is `dest`. Possible values for `datatype` are `MPI_INT`, `MPI_DOUBLE`, `MPI_CHAR` *etc.* `tag` is an integer used by the programmer to allow the receiver to select from among several arriving messages in the `MPI_Recv`. Finally, `comm` is something called a communicator, which is essentially a subset of the processes. Ordinarily, message passing occurs within a single subset. The subset `MPI_COMM_WORLD` consists of all the processes in a single parallel job, and is predefined.

A receive call matching the send above is

```
MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
          int tag, MPI_Comm comm, MPI_Status *status)
```

`buf` is where the data is placed once it arrives. `count`, an input argument, is the size of the buffer; the message is truncated if it is longer than the buffer. Of course, the receive has to be executed by the correct destination process, as specified in the `dest` part of the send, for it to match. `source` must be the rank of the sending process. The communicator and the tag must match. So must the datatype.

The purpose of the datatype field is to allow MPI to be used with heterogeneous hardware. A process running on a little-endian machine may communicate integers to another process on a big-endian machine; MPI converts them automatically. The same holds for different floating point formats. Type conversion, however, is not supported: an integer must be sent to an integer, a double to a double, *etc.*

Suppose the producer and consumer transact business in two word integer packets. The producer is process 0 and the consumer is process 1. Then the send would look like this:

```
int outgoing[2];
MPI_Send(outgoing, 2, MPI_INT, 1 100, MPI_COMM_WORLD)
```

and the receive like this:

```
MPI_Status stat;
int incoming[2];
MPI_Recv(incoming, 2, MPI_INT, 0 100, MPI_COMM_WORLD, &stat)
```

What if one wants a process to which several other processes can send messages, with service provided on a first-arrived, first-served basis? For this purpose, we don't want to specify the source in our receive, and we use the value `MPI_ANY_SOURCE` instead of an explicit source. The same is true if we want to ignore the tag: use `MPI_ANY_TAG`. The basic purpose of the status argument, which is an output argument, is to find out what the tag and source of such a received message are. `status.MPI_TAG` and `status.MPI_SOURCE` are components of the struct `status` of type int that contain this information after the `MPI_Recv` function returns.

This form of send and receive are "blocking", which is a technical term that has the following meaning. for the send, it means that `buf` has been read by the system and the data has been moved out as soon as the send returns. The sending process can write into it without corrupting the message that was sent. For the receive, it means that `buf` has been filled with data on return. (A call to `MPI_Recv` with no corresponding call to `MPI_Send` occurring elsewhere is a very good and often used method for hanging a message passing application.)

MPI implementations may use buffering to accomplish this. When send is called, the data are copied into a system buffer and control returns to the caller. A separate system process (perhaps using communication hardware) completes the job of sending the data to the receiver. Another implementation is to wait until a corresponding receive is posted by the destination process, then transfer the data to the receive buffer, and finally return control to the caller. MPI provides two variants of send, `MPI_Bsend` and `MPI_Ssend` that force the buffered or the rendezvous implementation. Lastly, there is a version of send that works in "ready" mode. For such a send, the corresponding receive must have been executed previously, otherwise an error occurs. On some systems, this may be faster than the blocking versions of send. All four versions of send have the same calling sequence.

NOTE: MPI allows a process to send itself data. Don't try it. On the SP-2, if the message is big enough, it doesn't work. Here's why. Consider this code:

```
if (myrank == 0)
   for(dest = 0; dest < size; dest++)
      MPI_Send(sendbuf+dest*count, count, MPI_INT, dest, tag, MPI_COMM_WORLD);
MPI_Recv(recvbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &stat);
```

The programmer is attempting to send data from process zero to all processes, including process zero; $4 \cdot \text{count}$ bytes of it. If the system has enough buffer space for the outgoing messages, this succeeds, but if it doesn't, then the send blocks until the receive is executed. But since control does not return from the blocking send to process zero, the receive never does execute. If the programmer uses buffered send, then this deadlock cannot occur. An error will occur if the system runs out of buffer space, however:

```
if (myrank == 0)
   for(dest = 0; dest < size; dest++)
      MPI_Bsend(sendbuf+dest*count, count, MPI_INT, dest, tag, MPI_COMM_WORLD);
MPI_Recv(recvbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &stat);
```

9.3.2 Tags and communicators

Tags are used to keep messages straight. An example will illustrate how they are used. Suppose each process in a group has one integer and one real value, and we wish to find, on process zero, the sum of the integers and the sum of the reals. Lets write this:

```
itag = 100;
MPI_Send(&intvar, 1, MPI_INT, 0, itag, MPI_COMM_WORLD);
ftag = 101;
MPI_Send(&floatvar, 1, MPI_FLOAT, 0, ftag, MPI_COMM_WORLD);
/****  Sends are done.   Receive on process zero   ****/
if (myrank == 0) {
   intsum = 0;
   for (kount = 0; kount < nprocs; kount++) {
      MPI_Recv(&intrecv, 1, MPI_INT, MPI_ANY_SOURCE, itag, MPI_COMM_WORLD, &stat);
      intsum += intrecv;
   }
   fltsum = 0;
   for (kount = 0; kount < nprocs; kount++) {
      MPI_Recv(&fltrecv, 1, MPI_FLOAT, MPI_ANY_SOURCE, ftag, MPI_COMM_WORLD, &stat);
      fltsum += fltrecv;
   }
}
```

It looks simple, but there are a lot of subtleties here! First, note the use of MPI_ANY_SOURCE in the receives. We're happy to receive the data in the order it arrives. Second, note that we use two different tag values to distinguish between the int and the float data. Why isn't the MPI_TYPE filed enough? Because MPI does not include the type as part of the message "envelope". The envelope consists of the source, destination, tag, and communicator, and these must match in a send-receive pair. Now the two messages sent to process zero from some other process are guaranteed to arrive in the order they were sent, namely the integer message first. But that does not mean that all of the integer message precede all of the float messages! So the tag is needed to distinguish them.

This solution creates a problem. Our code, as it is now written, sends off a lot of messages with tags 100 and 101, then does the receives (at process zero). Suppose we called a library routine written by another user before we did the receives. What if that library code uses the same message tags? Chaos results. We've "polluted" the tag space. Note, by the way, that synchronizing the processes before calling the library code does not solve this problem.

MPI provides communicators as a way to prevent this problem. The communicator is a part of the message envelope. So we need to change communicators while in the library routine. To do this, we use `MPI_Comm_dup`, which makes a new communicator with the same processes in the same order as an existing communicator. For example

```
void safe_library_routine(MPI_Comm oldcomm)
{
   MPI_Comm mycomm;
   MPI_Comm_dup(oldcomm, &mycomm);
   <library code using mycomm for communication>
   MPI_Comm_free(&mycomm);
}
```

The messages sent and received inside the library code cannot interfere with those send outside.

9.3.3 Who am I?

On the SP-2 and other multicomputers, one usually writes one program which runs on all the processors. In order to differentiate its behavior, (like producer and consumer) a process usually first finds out at runtime its rank within its process group, then branches accordingly. The calls

```
MPI_Comm_size(MPI_Comm comm, int *size)
```

sets `size` to the number of processes in the group specified by `comm` and the call

```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

sets `rank` to the rank of the calling process within the group (from 0 up to $n-1$ where n is the size of the group). Usually, the first thing a program does is to call these using `MPI_COMM_WORLD` as the communicator, in order to find out the answer to the *big* questions, "Who am I?" and "How many other 'I's are there?".

Okay, I lied. That's the second thing a program does. Before it can do anything else, it has to make the call

```
MPI_Init(int *argc, char ***argv)
```

where `argc` and `argv` should be pointers to the arguments provided by UNIX to main(). While we're at it, let's not forget that one's code needs to start with

```
#include "mpi.h"
```

The *last* thing the MPI code does should be

```
MPI_Finalize()
```

No arguments.

Here's an MPI multi-process "Hello World":

```
#include <stdio.h>
#include "mpi.h"

main(int argc, char** argv) {
  int i, myrank, nprocs;
  double a = 0, b = 1.1, c = 0.90;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  printf("Hello world!  This is process %d out of %d\n",myrank, nprocs);
  if (myrank == 0) printf("Some processes are more equal than others.");
  MPI_Finalize();
}  /*  main  */
```

which is compiled and executed on the SP-2 at Ames by

```
babbage1% mpicc -O3 example.c
babbage1% a.out -procs 2
```

and produces (on the standard output)

```
0:Hello world!  This is process 0 out of 2
1:Hello world!  This is process 1 out of 2
0:Some processes are more equal than others.
```

Another important thing to know about is the MPI wall clock timer:

```
double MPI_Wtime()
```

which returns the time in seconds from some unspecified point in the past.

9.3.4 Performance, and tolerance

Try this exercise. See how long a message of length n bytes takes between the call time the send calls send and the time the receiver returns from receive. Do an experiment and vary n. Also vary the rank of the receiver for a fixed sender. Does the model

$$\text{Elapsed_Time}(n, r) = \alpha + \beta n$$

work? (r is the receiver, and according to this model, the cost is receiver independent.)

In such a model, the latency for a message is α seconds, and the bandwidth is $1/\beta$ bytes/second. Other models try to split α into two components. The first is the time actually spent by the sending processor and the receiving processor on behalf of a message. (Some of the per-byte cost is also attributed to the processors.) This is called the overhead. The remaining component of latency is the delay as the message actually travels through the machines interconnect network. It is ordinarily much smaller than the overhead on modern multicomputers (ones, rather than tens of microseconds).

A lot has been made about the possibility of improving performance by "tolerating" communication latency. To do so, one finds other work for the processor to do while it waits for a message to arrive. The simplest thing is for the programmer to do this explicitly. For this purpose, there are "nonblocking" versions of send and receive in MPI and other dialects.

Nonblocking send and receive work this way. A nonblock **send start** call initiates a send but returns before the data are out of the send buffer. A separate call to **send complete** then blocks the sending process, returning only when the data are out of the buffer. The same two-phase protocol is used for nonblocking receive. The **receive start** call returns right away, and the **receive complete** call returns only when the data are in the buffer.

The simplest mechanism is to match a nonblocking receive with a blocking send. To illustrate, we perform the communication operation of the previous section using nonblocking receive.

```
MPI_Request request;
MPI_IRecv(recvbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &request);
if (myrank == 0)
   for(dest = 0; dest < size; dest++)
      MPI_Send(sendbuf+dest*count, count, MPI_INT, dest, tag, MPI_COMM_WORLD);
MPI_Wait(&request, &stat);
```

`MPI_Wait` blocks until the nonblocking operation identified by the handle `request` completes. This code is correct regardless of the availability of buffers. The sends will either buffer or block until the corresponding **receive start** is executed, and all of these will be.

Before embarking on an effort to improve performance this way, one should first consider what the payoff will be. In general, the best that can be achieved is a two-fold improvement. Often, for large problems, it's the bandwidth (the βn term) that dominates, and latency tolerance doesn't help with this. Rearrangement of the data and computation to avoid some of the communication altogether is required to reduce the bandwidth component of communication time.

9.3.5 Who's got the floor?

We usually think of send and receive as the basic message passing mechanism. But they're not the whole story by a long shot. If we wrote codes that had genuinely different, independent, asynchronous processes that interacted in response to random events, then send and receive would be used to do all the work. Now consider computing a dot product of two identically distributed vectors. Each processor does a local dot product of its pieces, producing one scalar value per processor. Then we need to add them together and, probably, broadcast the result. Can we do this with send and receive? Sure. Do we want to? No. No because it would be a pain in the neck to write and because the MPI system implementor may be able to provide the two necessary, and generally useful collective operations (sum, and broadcast) for us in a more efficient implementation.

MPI has lots of these "collective communication" functions. (And like the sum operation, they often do computation as part of the communication.)

Here's a sum operation, on doubles. The variable `sum` on process `root` gets the sum of the variables `x` on all the processes.

```
double x, sum;
int root, count = 1;
MPI_Reduce(&x, &sum, count, MPI_DOUBLE, MPI_SUM, root, MPI_COMM_WORLD);
```

The fifth argument specifies the operation; other possibilities are `MPI_MAX`, `MPI_LAND`, `MPI_BOR`, ... which specify maximum, logical AND, and bitwise OR, for example.

The semantics of the collective communication calls are subtle to this extent: nothing happens except that a process stops when it reaches such a call, until *all* processes in the specified group reach it. Then the reduction operation occurs and the result is placed in the `sum` variable on the *root* processor. Thus, reductions provide what is called a *barrier* synchronization.

There are quite a few collective communication operations provided by MPI, all of them useful and important. We will use several in the assignment. To mention a few, MPI_Bcast broadcasts a vector from one process to the rest of its process group; MPI_Scatter sends different data from one process to each process in its a group; MPI_Gather is the inverse of a scatter: one process receives and concatenates data from all processes in its group; MPI_Allgather is like a gather followed by a broadcast: all processes receive the concatenation of data that are initially distributed among them; MPI_Reduce_scatter is like reduce followed by scatter: the result of the reduction ends up distributed among the process group. Finally, `MPI_Alltoall` implements a very general communication in which each process has a separate message to send to each member of the process group.

Often the process group in a collective communication is some subset of all the processors. In a typical situation, we may view the processes as forming a grid, let's say a 2d grid, for example. We may want to do a reduction operation within rows of the process grid. For this purpose, we can use MPI_Reduce, with a separate communicator for each row.

To make this work, each process first computes its coordinates in the process grid. MPI makes this easy, with

```
int nprocs, myproc, procdims[2], myproc_row, myproc_col;
MPI_Dims_create(nprocs, 2, procdims);
myproc_row = myrank / procdims[1];
myproc_col = myrank % procdims[1];
```

Next. one creates new communicators, one for each process row and one for each process column. The calls

```
MPI_Comm my_prow, my_pcol;
MPI_Comm_split(MPI_COMM_WORLD, myproc_row, 0, &my_prow);
MPI_Comm_split(MPI_COMM_WORLD, myproc_row, 0, &my_prow);
```

create them and

```
MPI_Comm_free(&my_prow);
MPI_Comm_free(&my_pcol);
```

free them. The reduce-in-rows call is then

```
MPI_Reduce(&x, &sum, count, MPI_DOUBLE, MPI_SUM, 0, my_prow);
```

which leaves the sum of the vectors `x` in the vector `sum` in the process whose rank in the group is zero: this will be the first process in the row. The general form is

```
MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm newcomm)
```

As in the example above, the group associated with `comm` is split into disjoint subgroups, one for every different value of `color`; the communicator for the subgroup that this process belongs to is returned in `newcomm`. The argument `key` determines the rank of this process within `newcomm`; the members are ranked according to their value of `key`, with ties broken using the rank in `comm`.

9.4 Message Passing in PVM

PVM is a popular message passing system that allows a user to treat a heterogeneous collection of networked machines as a single "parallel virtual machine". It was developed at Oak Ridge National Lab, beginning in 1989, and it continues to evolve and improve. It is freely available, easy to download and install, and it runs in user mode; any user can configure any set of machines on which he has accounts into a PVM using PVM. PVM handles all communication and synchronization, as well as format conversion for machines with different internal representations of integers, floating point values, and strings.

PVM allows the virtual machine to be configured dynamically: computers can be added to or deleted from the VM during the course of a computation. Very useful if someone switches off one of the machines! A PVM computation consists of a number of separate programs, called tasks, that use the PVM library to communicate. Tasks can be created and can drop out, also dynamically.

To obtain PVM and the documentation, one can use the web:

`http://www.epm.ornl.gov/pvm/pvm_home.html`

The book, *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing* by Al Geist Adam Beguelin Jack Dongarra Weicheng Jiang Robert Manchek, and Vaidy Sunderam, is the best reference on PVM. It too has a home page and can be downloaded:

`http://www.netlib.org/pvm3/book/pvm-book.html`

9.4.1 PVM Basics

Once one makes PVM, there are two relevant pieces of software. One is the pvm deamon (called pvmd). You start it on any machine that it has been compiled for, and it then allows you to give commands to add other machines to the current *configuration*, which is the virtual parallel machine. The user's command interface to PVM is called the PVM console, and there is also an X Windows version of it, XPVM.

Here is an example session, running at HP Labs. The machine will include two HP workstations:

```
hplpp3% pvm
new pvm shell
t40001
pvm> add hplpp4
1 successful
                    HOST     DTID
                  hplpp4    80000
pvm> conf
2 hosts, 1 data format
                    HOST     DTID     ARCH   SPEED
                  hplpp3    40000     HPPA    1000
                  hplpp4    80000     HPPA    1000
pvm>
```

The machine configuration commands can also be given by function calls from a running PVM application.

A PVM application is just a C or Fortran 77 program compiled and linked against the PVM library. Here is a sample:

```
#include <stdio.h>
#include "pvm3.h"

main()
{
        int cc, tid;
        char buf[100];

        printf("i'm t%x\n", pvm_mytid());

        cc = pvm_spawn("hello_other", (char**)0, 0, "", 1, &tid);

        if (cc == 1) {
                cc = pvm_recv(-1, -1);
                pvm_bufinfo(cc, (int*)0, (int*)0, &tid);
                pvm_upkstr(buf);
                printf("from t%x: %s\n", tid, buf);

        } else
                printf("can't start hello_other\n");

        pvm_exit();
        exit(0);
}
```

This example illustrates some of PVM's outstanding features. First, once begun, this task first finds its own PVM task identifier (or *tid*) by calling the PVM library function `pvm_mytid`.

Next, it starts another PVM task, leaving it up to pvmd to decide which actual processor to run it on. The other task's executable file is called `hello_other`, the source for which appears below. Its tid is kept in order to use it for communication, later. In the `if` construct, we see the basics of PVM message passing. In PVM, a send is a three step process. First, one allocates a buffer using `pvm_initsend`; next, one packs data into the buffer using `pvm_pk*` routines, which specify datatype. When there are multiple machines in the configuration, this is where data format conversions occur. Last, afters its contents are all packed in, the buffer is sent with `pvm_send`, which gives the destination tid and a message tag. The receive is a similar process. The `pvm_recv` (which allows specification of source tid and message tag) returns a receive buffer identifier. This is then used to unpack the data with `pvm_upk*`.

Here is the code for `hello_other`. Notice the call to `pvm_parent` which returns the tid of the task that spawned this one.

```
#include "pvm3.h"

main()
{
        int ptid;
        char buf[100];
```

```
        ptid = pvm_parent();

        strcpy(buf, "hello, world from ");
        gethostname(buf + strlen(buf), 64);

        pvm_initsend(PvmDataDefault);
        pvm_pkstr(buf);
        pvm_send(ptid, 1);

        pvm_exit();
        exit(0);
}
```

After compiling and linking both programs, I can log into either of the two workstations and start `hello` from the UNIX prompt. Also, using the PVM console, I configure the two machines as above. The result is this:

```
hplpp4% hello
i'm t80001
from t40002: hello, world from hplpp3
hplpp4%
```

The ability to configure heterogeneous networks and use them as a parallel processor is a very impressive capability. It's what makes PVM so popular. As a message passing dialect, however, PVM lacks the power and sophistication of MPI: it has only one send mode, no asynchronous receive, no communication contexts, and limited collective communication functions. Thus, we expect MPI and PVM to evolve towards one another, each importing the strengths of the other in time.

Performance is another issue. PVM is a user application, and as such it is forced to use the communication facilities of UNIX, which add latency and limit bandwidth. Vendor supported MPI implementations appear to have significantly better performance at this time.

9.5 More on Message Passing

9.5.1 Nomenclature

- Performance:

  Latency: the time it takes to send a zero length message (overhead)
  Bandwidth: the rate at which data can be sent

- Synchrony:

  Synchronous: sending function returns when a matching receiving operation has been initiated at the destination.
  Blocking: sending function returns when the message has been safely copied.
  Non-blocking (asynchronous): sending function returns immediately. Message buffer must not be changed until it is safe to do so

- Miscellaneous:

  Interrupts: if enabled, the arrival of a message interrupts the receiving processor
  Polling: the act of checking the occurrence of an event
  Handlers: code written to execute when an interrupt occurs
  Critical sections: sections of the code which cannot be interrupted safely
  Scheduling: the process of determining which code to run at a given time
  Priority: a relative statement about the ranking of an object according to some metric

9.5.2 The Development of Message Passing

- Early Multicomputers:
  UNIVAC
  Goodyear MPP (SIMD.)
  Denelcor HEP (Shared memory)

- The Internet and Berkeley Unix (High latency, low bandwidth)

  Support for communications between computers is part of the operating system (sockets, ports,remote devices)
  Client/server applications appear

- Parallel Machines (lowest latency,high bandwidth - dedicated networks)

  Ncube
  Intel (Hypercubes, Paragon)
  Meiko (CS1,CS2)
  TMC (CM-5)

- Workstation clusters (lower latency, high bandwidth, popular networks)

  IBM (SP1,2) - optional proprietary network

- Software (libraries):

  CrOS, CUBIX, NX, Express
  Vertex
  EUI/MPL (adjusts to the machine operating system)
  CMMD
  PVM (supports heterogeneous machines; dynamic machine configuration)
  PARMACS, P4, Chamaleon
  MPI (analgum of everything above and adjusts to the operating system)

- Systems:

  Mach
  Linda (object-based system)

9.5.3 Machine Characteristics

- Nodes:

  CPU, local memory,perhaps local I/O

- Networks:

  Topology: Hypercube,Mesh,Fat-Tree, other
  Routing: circuit, packet, wormhole, virtual channel random
  Bisection bandwidth (how much data can be sent along the net)
  Reliable delivery
  Flow Control
  "State" : Space sharing, timesharing, stateless

- Network interfaces: Dumb: Fifo, control registers
  Smart: DMA (Direct Memory Access) controllers
  Very Smart: handle protocol managememt

9.5.4 Active Messages

[Not yet written]